

REPLAY INSTRUCTION MORPHING

BACKGROUND

5 1. Field

The present disclosure pertains to the field of processors. More particularly, the present disclosure pertains to a processor that may alter, transform, mutate, or otherwise “morph” instructions when difficulties are encountered during one or more initial
10 attempts to execute such instructions.

2. Description of Related Art

Improving the performance of computers or other processing systems generally improves overall throughput and/or provides a better user experience. Such improved
15 system performance may be achieved by increasing the rate at which instructions for the system are processed by a processor. Accordingly, it is desirable to produce advanced processors with improved instruction throughput.

Continuing to increase the performance of a processor, however, is a difficult task. Prior art processors already employ techniques of branch prediction, speculative execution,
20 and out-of-order (OOO) execution. Additionally, such processors typically include multiple parallel execution units to process numerous instructions in parallel. As increasing amounts of parallel hardware are employed, providing sufficient instructions to keep this hardware busy becomes increasingly difficult due to limited instruction level parallelism which may be extracted or due to instruction dependencies present in many

existing software programs.

Multi-threading is one technique that may be used to reduce idle time for parallel execution units. Multi-threading allows multiple programs or threads to share hardware resources. Due to the separate program sequences being executed, there is less likelihood of instruction dependencies seriously reducing execution unit utilization. Such multi-threaded machines inherently benefit from the additional parallelism resulting from executing multiple threads as long as multiple threads can be extracted or are explicitly provided by the software being executed.

Thus, large amounts of effort in designing modern processors have been applied to such instruction-dispatch focused techniques. These techniques at least in part strive to increase the number of instructions dispatched to the intended execution units. At times, however, significant latency-causing problems are encountered post-dispatch (e.g., faults, numeric computation problems, cache misses, etc.). An execution unit in a prior art processor is generally "stuck" with the instruction it got once the instruction has been dispatched to the execution unit.

Instruction decoding is a type of an alteration of an instruction that occurs after an instruction is received by a processor. Instruction decoding, however, generally involves expanding an instruction into microinstructions, or changing the encoding of an instruction into a more convenient form or another instruction set for execution by an execution unit. Instruction decoding does not generally go beyond a particular mapping of an input instruction to either individual signals or individual microinstructions. Moreover, instruction decoding is an inherently front-end operation in processing systems and lacks the ability to incorporate information gleaned throughout execution of an instruction.

Thus, prior art processors generally do not morph original instructions into altered instructions that execute more efficiently or otherwise differently than the original instructions once attempted execution has occurred.

0916108567 1063000

Brief Description of the Figures

The present invention is illustrated by way of example and not limitation in the
5 figures of the accompanying drawings.

Figure 1a illustrates one embodiment of a processor employing an instruction
morphing circuit.

Figure 1b illustrates techniques for morphing instructions which may be
employed by the system of Figure 1a.

10 Figure 2 illustrates one embodiment of a technique for morphing load instructions
when a cache miss occurs in a cache memory.

Figures 3a-3d illustrate various embodiments of techniques for dealing with
instruction dependencies using instruction morphing.

15 Figure 4 illustrates another embodiment of a system that utilizes disclosed
instruction morphing techniques.

Figure 5 illustrates one embodiment of a technique for handling page faults using
instruction morphing.

Figure 6 illustrates one embodiment of a technique for handling indirect
instructions such as indirect load instructions using instruction morphing.

20 Figure 7 illustrates one embodiment of a system that uses instruction morphing in
conjunction with numerical processing.

Figure 8 illustrates one embodiment of techniques for handling certain rare data
dependent mathematical operations.

Figure 9 illustrates various design representations or formats for simulation, emulation, and fabrication of a design using the disclosed techniques.

5

Detailed Description

The following description provides replay instruction morphing. In the following description, numerous specific details such as types of original and morphed instructions, circumstances under which morphing may be appropriate, system environments in which morphing may be embodied, execution unit and morphing circuitry interactions, and logic partitioning/integration choices are set forth in order to provide a more thorough understanding of the present invention. It will be appreciated, however, by one skilled in the art that the invention may be practiced without such specific details. In other instances, control structures and gate level circuits have not been shown in detail in order not to obscure the invention. Those of ordinary skill in the art, with the included descriptions, will be able to implement appropriate logic circuits without undue experimentation.

The presently disclosed instruction morphing techniques may advantageously allow more efficient execution of instructions in a processing system. By morphing certain instructions when particular hardware is unavailable or when proper completion is otherwise recognized as not being presently possible, the processor may free resources for use in performing other tasks.

One embodiment of a processor that performs instruction morphing is shown in Figure 1a. The processor of Figure 1a includes an execution unit 125 which receives instructions from a multiplexer 115. A checker 150 is coupled to the execution unit 125 and determines whether instructions have executed properly. Additional checkers and/or execution units may be added in some embodiments. Furthermore, a staging queue (not

shown) may receive instructions from the multiplexer 115 and pass the instructions to the checker 150 for checking in due course. Properly executed instructions are forwarded on to retirement, whereas improperly executed instructions are fed back to the multiplexer 115 for re-execution.

5 There are two ways an instruction can be fed back to the execution unit 125 from the checker 150. Morphing logic 120 is coupled to receive original instructions from the checker 150, and may detect a condition that warrants morphing of the instruction. In some cases, however, morphing is inappropriate. Therefore, the morphing logic 120 may return either the original instructions or morphed instructions to the multiplexer 115. In
10 some embodiments, certain types of instructions or specific instructions may be automatically morphed by the morphing logic 120. In other embodiments, certain conditions may cause morphing logic 120 to perform morphing operations. Various delays may be introduced or conditions tested prior to instructions being returned to the execution unit 125.

15 Figure 1b illustrates two techniques for morphing instructions which may be employed by the system of Figure 1a. At block 160, the processor attempts execution of the original instruction. The original instruction is received from the RECEIVED INSTRUCTIONS input of the multiplexer 115, and then passed to the execution unit to accomplish the execution indicated in block 160. The received instructions may be
20 received from various decoding, caching, or other front-end processing logic.

As indicated in block 165, a problem preventing successful present execution of the original instruction is detected. In the embodiment of Figure 1a, this detection is accomplished by the checker 150. If the instruction and/or the conditions indicate that

the instruction should be replayed without alteration, the morphing logic 120 may return the instruction to the multiplexer 115 without change.

If the instruction and/or the conditions indicate that the instruction should be morphed, the morphing logic 120, as indicated in block 170, alters the instruction so that it will execute more efficiently or at least differently. Various embodiments of specific morphing operations will be discussed below. As indicated in block 175, the morphed instruction is then executed.

In some cases, the morphed instruction is intended to replace the original instruction. In this case, as indicated in block 180, retirement of the morphed instruction completes the execution which was expected from the original instruction. In this case, the morphed instruction is a substitute instruction which produces the same results as the original instruction; however, the morphed instruction was at the time perceived to be a better or more efficient way of achieving those results.

In other cases, an instruction may be morphed to satisfy a precondition to the original instruction's successful execution. For example, a memory access may cause a page fault to occur. A precondition to proper execution of the memory access is that the page fault be resolved. A particular precondition may require numerous morphing operations to satisfy. Therefore, as indicated in block 185, the processor checks to determine whether the precondition is satisfied by the execution of a morphed instruction. If not, further morphing may be performed as indicated by the return to block 170. If the precondition is satisfied, the original instruction may be restored as indicated in block 190. Thereafter, the original instruction may be retired as indicated in block 195. In some cases, a replacement instruction may be executed instead of the original instruction

once the precondition has been satisfied.

Figure 2 illustrates one embodiment of a technique for morphing load instructions when a cache miss occurs in a cache memory. As indicated in block 200, the processor determines that a load instruction missed in a second level or above cache. By second level, it is meant the second lowest hierarchical cache, regardless of its particular label. In the embodiment of Figure 1a, the checker 150 receives a miss signal from the lowest level cache and therefore determines that the load instruction executed improperly. The morphing logic 120 receives signals (not shown) indicating that a higher level cache has also experienced a cache miss.

Under these conditions, it may be wasteful to continuously test the higher level cache as the load instruction circulates through the replay loop because the needed data will be written to both the higher level cache and the lowest level cache when retrieved. Therefore, bandwidth of the higher level cache may be saved by, subsequent to the higher level cache miss, only attempting to retrieve the data from the lower-level cache. Accordingly, as indicated in block 210, the load may be morphed to perform lookups only in the lowest level cache in subsequent iterations.

If valid data is found in the lowest level cache, as tested in block 215, then the load will execute properly. The original load may then be retired when the checker detects correct execution of the morphed load, as indicated in block 225. If valid data is not found in the lowest level cache, the morphed load instruction is replayed as indicated in block 220. While the load may continue to unsuccessfully execute a number of times, at least it does not wastefully consume bandwidth of the higher level cache in the process.

Figure 3a illustrates one embodiment of a technique for dealing with instruction

dependencies using instruction morphing. As indicated in block 300, a dependent instruction and the previous instruction on which depends are identified. In block 310, the inability to presently execute the previous instruction is detected. In the embodiment of Figure 1a, block 310 may be accomplished by the checker 150 detecting the erroneous
5 execution of the previous instruction.

Since the instruction on which the dependent instruction depends cannot be properly executed, it follows that the dependent instruction cannot presently be properly executed. Therefore, continuously attempting to execute the dependent instruction may be wasteful. Accordingly, as indicated in block 320, the dependent instruction may be
10 prevented from continuously executing by marketing the instruction as "poisoned". An instruction marked as "poisoned" is simply marked in a manner such that the replay system and/or the execution units recognize that execution of the instruction should not be attempted. For example, a valid bit may be suppressed so that the instruction appears to be invalid and therefore will not be executed. This technique may advantageously
15 reduce the number of unsuccessful attempts at executing a dependent instruction.

As indicated in block 330, a poison-clearing event is detected. A poison-clearing event either specifically indicates that the dependency has been resolved or is an event that could have caused the dependency to be resolved. In some embodiments, tracking the exact conditions which will cause each individual dependency to be resolved may be
20 prohibitively expensive. Therefore, common events which may cause particular dependencies to be resolved may be used to clear the poison indicators for one or more instructions. As indicated in block 330, the dependent instruction is marked as safe (not poisoned) for attempted execution.

Figure 3b illustrates one embodiment of the operations performed in block 330 of Figure 3a. In this embodiment, the detection of any instruction retiring is performed in block 332. As a result of the detection of the retirement of any instruction, all poisoned instructions are reset so that execution will again be attempted as indicated in block 334.

5 This embodiment is relatively inexpensive in terms of the hardware required for implementation; however, it may result in some undesirable execution of dependent instructions where the dependencies have not yet cleared.

Figure 3c illustrates another embodiment of the operations performed in block 330 of Figure 3a. In this embodiment, a write to a lowest level cache is detected in block 336. The detection of this write causes all poisoned bits for instructions to be reset as again indicated in block 334. This technique is also convenient in terms of the amount of hardware required, but may also result in some unnecessary execution of dependent instructions.

10

Figure 3d illustrates another embodiment of a technique for dealing with instruction dependencies using instruction morphing. Figure 3d shares blocks 300, 310, and 320 with Figure 3a. After a dependent instruction and a previous instruction on which it depends are identified in block 300, however, the embodiment of Figure 3d includes an additional operation. As indicated in block 305, the dependent instruction is tagged with an identifier that indicates the previous instruction on which it is dependent.

15

20 The identifier may be a sequence number of the instruction or any other value that serves to identify the previous instruction. Notably, the tagging performed in block 305 may be performed in a different sequence than the exact sequence shown in Figure 3d. For example, the dependent instructions may not be tagged until after one or both of blocks

310 and 320.

In block 336, the retirement of an instruction is detected. Since dependent instructions were earlier tagged with an indication of the instructions on which they depend, instruction-specific poison clearing may be performed. In other words, when an instruction retires, the poison indicators may be reset for only those instructions which depend on the retired instruction by comparing any tagged dependent instructions' indicators to the corresponding value for the instruction being retired. Thus, as indicated in block 338, the poison indication(s) for dependent instruction(s) with indicators that indicate the retired instruction are cleared. This technique may advantageously greatly reduce unnecessary execution of dependent instructions when their correct execution is precluded due to the fact that the instruction on which they are dependent has not yet been completed.

Figure 4 illustrates another embodiment of a system that utilizes instruction morphing techniques. The embodiment of Figure 4 includes execution logic 425 which receives instructions from a multiplexer 415. Additionally, a staging queue 410 receives instructions from the multiplexer 415. The staging queue 410 stores instructions dispatched to the execution logic 425 and passes such instructions on to a checker 450, which is also coupled to the execution logic 425, to determine whether the execution logic 425 has properly executed the instructions. As was the case in the embodiment of Figure 1a, the checker replays improperly executed instructions. Morphing logic 420 may morph instructions depending on the particular instruction and/or the conditions under which it improperly executed.

Also illustrated in Figure 4 is a page miss handler (PMH) 460 as well as a

translation lookaside buffer (TLB) 470 and a memory 480. According to known paging techniques, the system stores a number of page table entries in the TLB 470. When a page table entry is not found in TLB 470, a page walk is performed by the page miss handler 460 to retrieve the page descriptor entry (PDE) and subsequently the page table entry (PTE) from memory.

Figure 5 illustrates one embodiment of a technique for handling page faults in the system of Figure 4. As indicated in block 500, a page fault producing instruction is identified. This may be performed by the morphing logic 420 perhaps with inputs from the execution logic 425 and/or the checker 450. As indicated in block 510, the morphing logic 420 then morphs the page fault producing instruction into a load of a page descriptor entry for the page which caused the fault. If instruction morphing were not used to introduce the page descriptor entry load into the replay system, another instruction may be prevented from executing.

For example, instead of having the morphing logic 420 perform a morphing operation to retrieve the page descriptor entry, the page miss handler could insert a page descriptor entry load into the execution stream via the dashed connection 465. This newly added instruction would displace another instruction, causing the displaced instruction to circulate again through the replay system before it is given a chance to execute. Instead, since it is known that the page fault producing instruction can not successfully execute, it may be more efficient to morph that instruction rather than displacing another instruction which could potentially successfully execute in the interim.

Similarly, after the page descriptor entry load is completed, as indicated in block 520, the page descriptor entry load may be morphed into a page table entry load (block

530). Again, this morphing technique avoids displacing another instruction. As indicated in block 540, the page table entry load completes, and the instruction may be morphed back into the original page fault producing instruction (block 550). This instruction may now execute without causing a page fault, with the page fault being resolved without displacing other operations in the replay system.

Figure 6 illustrates one embodiment of a technique for handling indirect instructions. Indirect instructions such as MOV EBX, MEM[EAX] involve two retrievals. First, the value of EAX must be retrieved in order to find the address of the data which is requested to be loaded into EBX. Second, the actual memory access to the address (the contents of EAX) of the requested data is performed. If the instruction is decoded into multiple microoperations, then additional resources are consumed. Thus, it may be advantageous to have indirect instructions which are not decoded into multiple microoperations, but rather which are morphed to perform the proper operations.

Accordingly, the technique shown in Figure 6 may be used to allow a single microoperation to accomplish indirect addressing. In block 600, an indirect memory reference instruction is identified. Indirect memory referencing techniques and instructions are well known and will not be further discussed herein. This technique may be used for a variety of indirect or similar addressing techniques which implicitly require multiple memory or register accesses or a combination of memory and register accesses.

In block 610, the instruction is morphed into an altered instruction which loads the address of the requested data. In the above example (MOV EBX, MEM[EAX]), the value of EAX would be loaded and is received by the memory execution unit as indicated in block 620. Next, the instruction is morphed into a load of the requested data as

indicated in block 630, and a load of the memory location that was indicated by the EAX register is performed. Accordingly, a single instruction slot may be used to perform indirect or similar memory access techniques.

Figure 7 illustrates an embodiment of replay instruction morphing that deals with numerical computations. In the embodiment of Figure 7, an over-precise or high precision case is handled specially by the replay system. An over-precise or high precision case may be either an instruction or a particular data-dependent case which requires additional cycles or hardware to compute a result to the desired precision. Thus, the need for additional resources may be due to the precision requested by the instruction or the particular numbers involved.

In block 700, the over-precise or high precision case is detected. Instead of attempting to compute the final result, the execution unit computes an intermediate result as indicated in block 710. The instruction is morphed, as indicated in block 715 and then tagged as an over-precise replay. Next, the morphed or altered instruction using the intermediate result is executed, as indicated in block 720. The final result is placed into the proper destination location as indicated in block 725.

There may be several reasons why the computation of only an intermediate result is advantageous. In some cases, it may be possible to use simpler hardware that cannot compute results for all input data in the same number of cycles. Typically, some rare cases require significant additions to hardware to ensure proper handling in the same time frame as other numbers. In such cases, the correct final result may be obtained via replay, and hardware may be saved. Additionally, some different higher precision instructions may advantageously be handled in a manner similar to lower precision instructions,

except that they are passed back through the replay system to compute final and sufficiently accurate results.

Similarly, Figure 8 illustrates one embodiment of techniques for handling certain rare data dependent mathematical operations. In the embodiment of Figure 8, substitutions of instructions and/or operands may be performed to advantageously simplify hardware. Again, hardware otherwise needed to handle difficult and rare cases may be eliminated, resulting in a more compact part, with only rare impacts to performance.

As indicated in block 800, a data dependent computationally intensive or hardware intensive mathematical operation is detected. For example, certain round instructions are very computationally intensive and therefore require significant amounts of hardware. In block 810, the instruction is morphed into a less hardware and/or computation intensive operation. For example, a round operation may be morphed into an add instruction or a subtract instruction, depending on the exact operand involved.

Finally, the substitute operation is executed to produce the identical result as indicated in block 820. The result is "identical" to the user in that, to the precision requested, the result produced by the numerical execution unit is the same as would be produced if the original instruction had been performed. Thus, the user may be unaware that an add was performed instead of a round, but the execution unit itself may be simplified so that it need not handle rare and difficult cases.

Figure 9 illustrates various design representations or formats for simulation, emulation, and fabrication of a design using the disclosed techniques. Data representing a design may represent the design in a number of manners. First, as is useful in

simulations, the hardware may be represented using a hardware description language or another functional description language which essentially provides a computerized model of how the designed hardware is expected to perform. The hardware model 910 may be stored in a storage medium 900 such as a computer memory so that the model may be simulated using simulation software 920 that applies a particular test suite 930 to the hardware model 910 to determine if it indeed functions as intended. In some embodiments, the simulation software is not recorded, captured, or contained in the medium.

Additionally, a circuit level model with logic and/or transistor gates may be produced at some stages of the design process. This model may be similarly simulated, sometimes by dedicated hardware simulators that form the model using programmable logic. This type of simulation, taken a degree further, may be an emulation technique. In any case, re-configurable hardware is another embodiment that may involve a machine readable medium storing a model employing the disclosed techniques.

Furthermore, most designs, at some stage, reach a level of data representing the physical placement of various devices in the hardware model. In the case where conventional semiconductor fabrication techniques are used, the data representing the hardware model may be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. Again, this data representing the integrated circuit embodies the techniques disclosed in that the circuitry or logic in the data can be simulated or fabricated to perform these techniques.

In any representation of the design, the data may be stored in any form of a computer readable medium. An optical or electrical wave 960 modulated or otherwise

